Formal verification using System Verilog Assertions (SVA)

Christian Krieg

October 15, 2024

Welcome to this year's *Digital Integrated Circuits* course! Throughout this course, we will have to solve some assignments that will teach us fundamental concepts of integrated circuit design.

We will demonstrate digital integrated circuit design on a step-by-step simple example of a digital counter. As the name of this course suggests, our ultimate goal is to implement this counter as a digital integrated circuit. In the lab, we map this counter to a field-programmable gate array (FPGA) architecture, and run it on the board shown in Figure 1. The counter's functionality is executed on the FPGA. We will be able to control execution by pushing the buttons, and to visualize the counter value using the light-emitting diodes (LEDs). Because operating a hardware counter with buttons and LEDs may be inconvenient, we also will interface the hardware counter over a universal asynchronous receiver/transmitter (UART), so that we may be able to further process the counter value on a host computer. But this will all happen during the lab week at the end of the exercises.

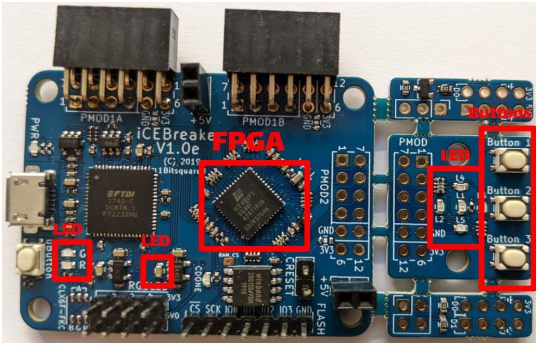

Figure 1: The *icebreaker* board, which will run our counter

Before we can bring our counter on hardware, it is important that we learn how to design digital circuits using state-of-the-art (and beyond) methods and tools, and to get a feeling for the results of our design decisions. We will draw attention on the results of the synthesis steps, which we will verify with several tools along the digital design flow.

Generally speaking, we perform the following steps:

1. **Specify the counter's behavior**
2. Model (implement) the counter in a hardware description language (HDL)
3. Verify the counter's implementation
4. Synthesize and optimize the counter's HDL model + verify
5. Map the counter to the target (FPGA) technology + verify
6. Generate an FPGA bitstream + verify
7. Configure the FPGA with the bitstream + verify
8. Run the counter on hardware

Our overall task is to build a synchronous counter that runs in two modes, based on a *mode* switch. The two modes are called *up* and *down*. In *up* mode, the counter increments the counter value by **5** at the rising edge of the clock signal, and in *down* mode, it decrements the counter value by **9** at the rising edge of the clock signal. The counter should implement synchronous reset behavior, such that it is set to an initial value of **-50** upon reset. The counter should never exceed a value of **235**, and should never go below a value of **-230** (the counter value sticks near the minimum/maximum value, until the counting direction changes). The counter value should never be **-11** (the counter should jump over this value by one increment/decrement).

Wrapping up: Simulation-based verification

In the previous task, we learned that we may use an HDL simulator to verify the correctness of a digital design. The simulator takes the design under test (DUT) as input, as well as test input values and assertions in order to check whether the design does what it should do.

We learned that it may be tedious to find sequences of input values that stimulate the DUT such that it enters a potential error state. The big lesson we learn is that the quality of our verification results are determined by the quality and quantity of assertions AND test input values.

Basically, what we have to do is:

1. Formulate assertions that verify correct increment/decrement values for all modes of operations
2. Generate an input value sequence to approach the minimum such that the counter value lands on a value close to, but not exactly at the minimum (let's say $\mathrm{MIN} + 1$)
3. Generate an input value sequence approach the maximum such that the counter value lands on a value close to, but not exactly at the maximum (let's say $\mathrm{MAX} - 1$)
4. Generate an input value sequence stimulate the counter such that it jumps over the invalid value while counting up
5. Generate an input value sequence stimulate the counter such that it jumps over the invalid value while counting down

Formulating assertions may be accomplished in a fairly systematic, straight-forward fashion.

Finding input value sequences that stimulate the DUTs may be a bit trickier. While we could follow a trial-and-error approach to find valid sequences in this simple example, this may not be a feasible approach for real-world designs.

Having that said, we may calculate any target counter value by starting from the initial value, and counting x times up, and y times down.

$$\mathrm{TARGET} = \mathrm{INIT} + x \cdot \mathrm{INC} - y \cdot DEC \tag{1}$$

Then, we may transform Equation (1) such that

$$y = \frac{\mathrm{INIT} - \mathrm{TARGET}}{\mathrm{DEC}} + x \cdot \frac{\mathrm{INC}}{\mathrm{DEC}}. \tag{2}$$

Then, we may select x and y such that we stimulate the counter as desired, while satisfying all its properties.

For the example counter that we used in the tutorial of the previous task, we may come up with a testbench as given in Listing 1. The example counter has the parameters as given in Table 1:

Table 1: Parameters for the example counter

Parameter	Value
MIN	-287
MAX	309
INV	75
INIT	13
INC	4
DEC	10

The assertions that check for correct values of the counter's increment and decrement are given in Lines 109 to 115 and Lines 117 to 123. In Lines 56 to 81, we generate the input sequences with respect to Equation (2), in order to test the counter's behavior at the specified edge cases. Please note that we let the counter count one step more as calculated in Equation (2), such that potential violations of the specification may be able to occur (Lines 60, 67, 74 and 81).

Listing 1: Sample solution for the testbench given in file *tb_counter.v*

```
1 `timescale 1ns/1ns
2
3 `ifndef VCDFILE
4 `define VCDFILE "counter.vcd"
5 `endif
6
```

```
`define myassert(x) \
   assert(x) \
   else begin \
      $error("FAIL: assert(%s) in %s:%0d", `"x`", `__FILE__, `__LINE__); \
      $finish_and_return(1);\
   end

module tb_counter;

   // Specify dump file and the scope to be dumped
   initial begin
      $dumpfile(`VCDFILE);
      $dumpvars(0, tb_counter);
   end

   // Define inputs
   reg clk=0;
   reg rst=0;
   reg mode=0;

   // Define outputs
   wire signed [9:0] cnt;

   // Instantiate design under test (DUT)
   counter dut (
      .clk(clk),
      .rst(rst),
      .mode(mode),
      .cnt(cnt)
   );

   // Generate clock
   always #5 clk = !clk;

   // Stimulate the design (vary input values)
   initial begin

      // Reset the DUT
      #5 rst = 1;
      #10 rst = 0;

      // Provide some inputs
      #17 mode = !mode;
      #22 mode = !mode;
      #12 mode = !mode;
      #100 mode = !mode;
      #1000 mode = !mode;

      // Navigate to MAX (counting up)
      #5 mode = 0;
      #5 rst=1;
      #10 rst = 0;
      #(10*3) mode=1; // y=3
      #(10*82) ; // x=81

      // Navigate to MIN (counting down)
      #5 mode = 1;
      #5 rst=1;
      #10 rst = 0;
      #(10*1) mode=0; // x=1
      #(10*31); // y=30

      // Now navigate to invalid value (counting up)
      #5 mode = 0;
      #5 rst=1;
      #10 rst = 0;
      #(10*3) mode = 1; // y=3
      #(10*24); // x=23

      // Now navigate to invalid value (counting down)
      #5 mode = 1;
      #5 rst=1;
      #10 rst = 0;
      #(10*23) mode = 0; // x=23
      #(10*4); // y=3

      $finish;

   end

   initial $monitor("At time %0t, value = %h (%0d)", $time, cnt, cnt);

   // Store past values of signals rst, mode, and cnt
   reg signed [9:0] past_cnt = 'x;
   reg past_mode = 'x;
   reg past_rst = 'x;
   always @(posedge clk) begin
      past_cnt <= cnt;
      past_rst <= rst;
      past_mode <= mode;
   end
```

```
98
99    // Verify properties of the counter
100   always @(posedge clk) begin
101      `myassert (cnt >= -287);
102      `myassert (cnt <= 309);
103      `myassert (cnt != 75);
104      if(past_rst) begin
105         `myassert(cnt == 13);
106      end else
107      if (!rst) begin
108         if (past_mode == 1) begin
109            if(past_cnt==(75-4)) begin // counter jumped over INV
110               `myassert((cnt-past_cnt)==(4+4));
111            end else if (past_cnt<=309 && past_cnt>(309-4)) begin // counter near MAX
112               `myassert((cnt-past_cnt)==0);
113            end else begin
114               `myassert((cnt-past_cnt)==4);
115            end
116         end else begin // past_mode == 0
117            if(past_cnt == (75+10)) begin // counter jumped over INV
118               `myassert((past_cnt-cnt)==(10+10));
119            end else if (past_cnt>=-287 && past_cnt<(-287 + 10)) begin // counter near MAX
120               `myassert((past_cnt-cnt)==0);
121            end else begin
122               `myassert((past_cnt-cnt)==10);
123            end
124         end
125      end
126   end
127 endmodule
```

Formal verification

As we just learned, it may be challenging to design testbenches that verify correct behavior of a DUT, and that may capture potentially incorrect behavior. We have to assert all the DUT's specified properties, and we have to generate inputs to the DUT that may trigger potential error cases.

This is where formal verification comes in very handy. In formal verification (or, more specifically, in *model checking*), we may specify properties of a DUT just as we did in our simulation testbench. A so-called *model checker* reads the DUT, and checks whether these properties hold **for all possible input values (and sequences)**. In other words: The model checker tries to find input valuations which violate any of the specified properties.

This is fairly cool. When using a model checker, the only thing we have to do is to specify properties. The tedious task of finding inputs is done automatically, which brings huge improvements in (testbench) design time, and, more importantly, in verification coverage.

There are many tools available for formal verification. In this course, we use *SystemVerilog Assertions (SVA)* to specify the properties of our counter (they are widely used in industry), and *SymbiYosys*[1] for checking our counter against these properties (it is freely accessible – it's open-source). While there is an open-source version of SymbiYosys available, we use a proprietary extension in order to leverage features which are not available in the open-source version (i.e., the *bind* operator, see below).

A testbench for formal verification

Just like we did in simulation-based verification, we may define a testbench in formal verificaion. While the main structure is equivalent, there are slight differences, which we will discuss now.

Listing 2 shows an example testbench, implemented in the file *tb_counter.sv*. Please note the file extension *sv*, which stands for *SystemVerilog*. The file hosts two modules: *tb_counter* (Lines 1 to 19) and *assertions* (Lines 22 to 63). Please note that there is no *timescale* directive in the formal testbench.

We encapsulate all assertions which check our counter's properties in the *assertions* module. Other than in simulation-based verification, we do not specify the progress of time (or, *delay*) by making use of the hash operator. Instead, we relate all behavior to a global clock signal, and specify all properties depending on this clock signal. The clock signal's values are not generated by us, but is provided by the model checker. Lines 23 to 26 shows the port definiton of the *assertions* module. We notice that there are only inputs, and no outputs. This is because the *assertions* module implements code to check our counter, taking all its inputs and outputs for verifying its correct behavior.

In Lines 30 to 35, we generate a reset sequence. We notice *assume* statements, which is used to constrain signal values. With *assume*, we take control of assigning values to a signal, and let the model checker know that it only may vary this signal according to the condition within the *assume* statement to find input valuations that violate any property.

[1] https://symbiyosys.readthedocs.io/en/latest/

The *always* block defined in Lines 38 to 62 finally hosts all the assertions we use to check the properties of our counter. We may find this block looking very familiar. Two assertions are given: In Line 41, the testbench checks if the counter value is set to the initial value upon reset. In Line 50, the testbench checks whether the counter value is always below or equal to the specified maximum value.

The *tb_counter* (Lines 1 to 19) module finally glues everything together. In Lines 9 to 14, we instantiate the DUT (a counter), and in Line 17 we connect the DUT to the *assertions* module using the *bind* statement. This way, it is possible to keep design implementation separated from verification code, which may be required to separate work between a design team and a verification team.

Please note that – other than in simulation-based verification – the *tb_counter* testbench module has ports connecting to the outside world. This is necessary for the model checker to access the testbench, and to find input values that provoke some output values that violate some property.

Listing 2: A formal testbench for our counter, implemented in the file *tb_counter.sv*

```
1  module tb_counter (
2        input clk,
3        input rst,
4        input mode,
5        output signed[9:0] cnt
6  );
7
8     // Instantiate DUT
9     counter dut(
10       .clk(clk),
11       .rst(rst),
12       .mode(mode),
13       .cnt(cnt)
14    );
15
16    // Connect DUT with assertions
17    bind dut assertions tb (.*);
18
19 endmodule
20
21
22 module assertions (
23       input clk,
24       input rst,
25       input mode,
26       input signed[9:0] cnt
27 );
28
29    // Generate reset sequence
30       reg init = 1;
31       always @(posedge clk) begin
32             if (init) assume(rst);
33             else assume(!rst);
34             init <= 0;
35       end
36
37    // Check properties
38       always @(posedge clk) begin
39
40             if (rst) begin
41                   assert (cnt == 13);
42             end
43
44             if (!rst) begin
45
46                   //
47                   // Check if counter value is never lower than MIN, larger than MAX, or
48                   // equal to INV
49                   //
50                   assert (cnt <= 309);
51
52                   //
53                   // Check if the counter value is correctly incremented and decremented
54                   //
55
56                   // Counting up
57          // TODO: Add your properties here!
58
59                   // Counting down
60          // TODO: Add your properties here!
61             end
62       end
63 endmodule
```

Configuring formal verification tasks

As already outlined, we use SymbiYosys to formally verify our counter. Of course, we still do not have an implementation of our counter. We will implement one soon, but in the meanwhile we use the test counters provided with this task.

SymbiYosys takes as input a configuration file that specifies the verification tasks. Listing 3 shows the configuration file we will use in this task. It may appear overwhelming at the first glimpse, but we will examine each section step by step, and we may find SymbiYosys's configuration system being pretty clever.

The configuration file hosts five sections: *tasks*, *options*, *engines*, *script*, and *files*.

In the *tasks* section (Lines 1 to 12), we may define independent tasks, which may be used in the other sections to selectively provide configurations for the different tasks. We define eleven tasks; one for each test counter provided with this task.

In the *options* section (Lines 14 to 15), we tell the model checker to operate in *prove* mode, which means that it tries to find property violations for all possible input valuations and an unbounded number of clock cycles. A different mode would be *bmc*, which checks properties up to a pre-defined number of clock cycles, specified via a *depth* option. Due to computational complexity, it may be necessary to introduce this sort of limitation for large designs in order to let the model checker terminate within a reasonable time span.

In the *engines* section (Lines 17 to 18), we may select specific model checkers and solvers (satisfiability modulo theories (SMT) solver and/or satisfiability (SAT) solvers). In our example, we select the *Aiger* model checker using the *suprove* solver, which supports unbounded model checking.

In the *script* section (Lines 20 to 35), we define the Yosys script that processes our design along with its testbench. *Yosys* is the synthesis tool that we use in this course. We basically read all the source files (depending on the verification task, we read a different test counter), and prepare it for the model checker.

Finally, the *files* section lists all the files (along with their paths) needed for the verification tasks. While the list of files may be self-explaining, we use a very practical feature of SymbiYosys in Lines 50 to 53, which lets us define configuration lines using Python. As our test counters need a cell library *simcells.v* to be fully understood by SymbiYosys, we use Python to call the utility *yosys-config*, with which we determine the path Yosys installed this cell library on the system where we run the verification.

Listing 3: *SymbiYosys* configuration file *counter.sby*

```
1  [tasks]
2  0
3  1
4  2
5  3
6  4
7  5
8  6
9  7
10 8
11 9
12 10
13
14 [options]
15 mode prove
16
17 [engines]
18 aiger suprove
19
20 [script]
21 read -formal simcells.v
22 read -formal tb_counter.sv
23 0: read -formal counter-0.v
24 1: read -formal counter-1.v
25 2: read -formal counter-2.v
26 3: read -formal counter-3.v
27 4: read -formal counter-4.v
28 5: read -formal counter-5.v
29 6: read -formal counter-6.v
30 7: read -formal counter-7.v
31 8: read -formal counter-8.v
32 9: read -formal counter-9.v
33 10: read -formal counter-10.v
34 verific -import tb_counter
35 prep -top tb_counter
36
37 [files]
38 tb_counter.sv
39 0: tests/counter-0.v
40 1: tests/counter-1.v
41 2: tests/counter-2.v
42 3: tests/counter-3.v
```

```
4: tests/counter-4.v
5: tests/counter-5.v
6: tests/counter-6.v
7: tests/counter-7.v
8: tests/counter-8.v
9: tests/counter-9.v
10: tests/counter-10.v
--pycode-begin--
import subprocess
output(subprocess.run(["yosys-config", "--datdir/simcells.v"], capture_output=True).stdout.decode("utf-8"))
--pycode-end--
```

Running the formal verification

Now that we have configured the verification tasks, it is time for us to actually formally verify our test counters. In general, a call to *SymbiYosys* specifies the configuration file, as well as the verification task(s) to be performed. If no tasks are given, all verification tasks are performed (Listing 4).

Listing 4: General call to *SymbiYosys*

```
sby <configuration-file> [task(s)]
```

So, let us first examine the correctly implemented counter (Listing 5). The `-f` option forces *SymbiYosys* to over-write any existing output directory from a previous verification run. We notice that the model checker returns a *PASS* in Line 28.

Listing 5: Formally verifying the correct counter

```
$ sby -f counter.sby 0
SBY 00:00:01 [counter_0] Removing directory '<path>/counter_0'.
SBY 00:00:01 [counter_0] Copy '<path>/tb_counter.sv' to '<path>/counter_0/src/tb_counter.sv'.
SBY 00:00:01 [counter_0] Copy '<path>/tests/counter-0.v' to '<path>/counter_0/src/counter-0.v'.
SBY 00:00:01 [counter_0] Copy '<path>/.local/opt/tabby/share/yosys/simcells.v' to '<path>/counter_0/src/simcells.v'.
SBY 00:00:01 [counter_0] engine_0: aiger suprove
SBY 00:00:01 [counter_0] base: starting process "cd counter_0/src; yosys -ql ../model/design.log ../model/design.ys"
SBY 00:00:01 [counter_0] base: Academic/educational license for ICT / TU-Vienna.
SBY 00:00:01 [counter_0] base: Do not redistribute without permission.
SBY 00:00:01 [counter_0] base: Do not use for commercial purposes.
SBY 00:00:01 [counter_0] base: finished (returncode=0)
SBY 00:00:01 [counter_0] prep: starting process "cd counter_0/model; yosys -ql design_prep.log design_prep.ys"
SBY 00:00:01 [counter_0] prep: Academic/educational license for ICT / TU-Vienna.
SBY 00:00:01 [counter_0] prep: Do not redistribute without permission.
SBY 00:00:01 [counter_0] prep: Do not use for commercial purposes.
SBY 00:00:01 [counter_0] prep: finished (returncode=0)
SBY 00:00:01 [counter_0] aig: starting process "cd counter_0/model; yosys -ql design_aiger.log design_aiger.ys"
SBY 00:00:01 [counter_0] aig: Academic/educational license for ICT / TU-Vienna.
SBY 00:00:01 [counter_0] aig: Do not redistribute without permission.
SBY 00:00:01 [counter_0] aig: Do not use for commercial purposes.
SBY 00:00:02 [counter_0] aig: finished (returncode=0)
SBY 00:00:02 [counter_0] engine_0: starting process "cd counter_0; suprove model/design_aiger.aig"
SBY 00:00:02 [counter_0] engine_0: finished (returncode=0)
SBY 00:00:02 [counter_0] engine_0: Status returned by engine: PASS
SBY 00:00:02 [counter_0] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 00:00:02 [counter_0] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:01 (1)
SBY 00:00:02 [counter_0] summary: engine_0 (aiger suprove) returned PASS
SBY 00:00:02 [counter_0] DONE (PASS, rc=0)
```

Now, let us formally verify the counter that implements an incorrect maximum value (Listing 6). Line 50 tells us that the model checker return with *FAIL*. By examining the log output, we find out that an assertion failed (Line 39), that a counterexample trace is generated (Line 49) which shows valuations of all of the DUT's signals that violate the property checked by the assertion. A look at Line 50 in Listing 2 reveals that the property which checks the correct maximum value failed in Listing 6.

Listing 6: Formally verifying the counter with incorrect maximum value

```
$ sby -f counter.sby 1
SBY 00:00:04 [counter_1] Removing directory '<path>/counter_1'.
SBY 00:00:04 [counter_1] Copy '<path>/tb_counter.sv' to '<path>/counter_1/src/tb_counter.sv'.
SBY 00:00:04 [counter_1] Copy '<path>/tests/counter-1.v' to '<path>/counter_1/src/counter-1.v'.
SBY 00:00:04 [counter_1] Copy '<path>/.local/opt/tabby/share/yosys/simcells.v' to '<path>/counter_1/src/simcells.v'.
SBY 00:00:04 [counter_1] engine_0: aiger suprove
SBY 00:00:04 [counter_1] base: starting process "cd counter_1/src; yosys -ql ../model/design.log ../model/design.ys"
SBY 00:00:04 [counter_1] base: Academic/educational license for ICT / TU-Vienna.
SBY 00:00:04 [counter_1] base: Do not redistribute without permission.
```

```
SBY 00:00:04 [counter_1] base: Do not use for commercial purposes.
SBY 00:00:04 [counter_1] base: finished (returncode=0)
SBY 00:00:04 [counter_1] prep: starting process "cd counter_1/model; yosys -ql design_prep.log
    design_prep.ys"
SBY 00:00:04 [counter_1] prep: Academic/educational license for ICT / TU-Vienna.
SBY 00:00:04 [counter_1] prep: Do not redistribute without permission.
SBY 00:00:04 [counter_1] prep: Do not use for commercial purposes.
SBY 00:00:04 [counter_1] prep: finished (returncode=0)
SBY 00:00:04 [counter_1] aig: starting process "cd counter_1/model; yosys -ql design_aiger.log
    design_aiger.ys"
SBY 00:00:04 [counter_1] aig: Academic/educational license for ICT / TU-Vienna.
SBY 00:00:04 [counter_1] aig: Do not redistribute without permission.
SBY 00:00:04 [counter_1] aig: Do not use for commercial purposes.
SBY 00:00:04 [counter_1] aig: finished (returncode=0)
SBY 00:00:04 [counter_1] engine_0: starting process "cd counter_1; suprove model/design_aiger.aig"
SBY 00:00:05 [counter_1] engine_0: finished (returncode=0)
SBY 00:00:05 [counter_1] engine_0: Status returned by engine: FAIL
SBY 00:00:05 [counter_1] smt2: starting process "cd counter_1/model; yosys -ql design_smt2.log
    design_smt2.ys"
SBY 00:00:05 [counter_1] smt2: Academic/educational license for ICT / TU-Vienna.
SBY 00:00:05 [counter_1] smt2: Do not redistribute without permission.
SBY 00:00:05 [counter_1] smt2: Do not use for commercial purposes.
SBY 00:00:05 [counter_1] smt2: finished (returncode=0)
SBY 00:00:05 [counter_1] engine_0: starting process "cd counter_1; yosys-smtbmc -s yices --noprogress
    --append 0 --dump-vcd engine_0/trace.vcd --dump-yw engine_0/trace.yw --dump-vlogtb engine_0/
    trace_tb.v --dump-smtc engine_0/trace.smtc --aig model/design_aiger.aim:engine_0/trace.aiw model/
    design_smt2.smt2"
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 Solver: yices
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 Skipping step 0 (and assuming pass)..
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 Skipping step 1 (and assuming pass)..
[...]
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 Skipping step 75 (and assuming pass)..
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 Skipping step 76 (and assuming pass)..
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 Checking assertions in step 77..
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 BMC failed!
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 Assert failed in tb_counter.dut.tb: tb_counter.sv:50 (
    $auto$verificsva.cc:1726:import$1009)
SBY 00:00:05 [counter_1] engine_0: ## 0:00:00 Writing trace to VCD file: engine_0/trace.vcd
SBY 00:00:07 [counter_1] engine_0: ## 0:00:02 Writing trace to Verilog testbench: engine_0/trace_tb.v
SBY 00:00:07 [counter_1] engine_0: ## 0:00:02 Writing trace to constraints file: engine_0/trace.smtc
SBY 00:00:07 [counter_1] engine_0: ## 0:00:02 Writing trace to Yosys witness file: engine_0/trace.yw
SBY 00:00:08 [counter_1] engine_0: ## 0:00:02 Status: FAILED
SBY 00:00:08 [counter_1] engine_0: finished (returncode=1)
SBY 00:00:08 [counter_1] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:03 (3)
SBY 00:00:08 [counter_1] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:05 (5)
SBY 00:00:08 [counter_1] summary: engine_0 (aiger suprove) returned FAIL
SBY 00:00:08 [counter_1] summary: counterexample trace: counter_1/engine_0/trace.vcd
SBY 00:00:08 [counter_1] DONE (FAIL, rc=2)
SBY 00:00:08 The following tasks failed: ['1']
```

Debugging the design

So, now that the formal verification failed, we need to investigate the root cause of the problem. We open the generated counterexample trace using *GTKWave* as shown in Listing 7. Please note the ampersand (&) at the end of the command line, which sends *GTKWave* to the background, so the shell remains usable while *GTKWave* runs.

Listing 7: Calling *GTKWave*

```
$ gtkwave counter_1/engine_0/trace.vcd &
```

Figure 2 shows *GTKWave*'s graphical user interface (GUI). We select *tb_counter* in the upper-left tree view, and add the signals *clk*, *rst*, *mode*, and *cnt* to the waveform diagram by double-clicking them in the lower-left list. In order to see the full trace, we click the *Zoom Fit* button in the bar on the top of the window. Finally, we right-click the *cnt* signal in the waveform diagram's list of signals, and select *Signed Decimal* as *Data Format*. Having a look at the *cnt* signal shows that its last value is 314, which is one increment larger than the specified maximum value. We may use this information to further debug the DUT's source code.

Key take-away

Please note that the model checker took over your job from the previous task to find a suitable timing and input valuation to exhibit incorrect behavior in the DUT. We did not manually provide any input valuation to the DUT.

Your task

As in the previous task, we again provide a ZIP archive which contains all necessary data to quickly solve this task. In addition to the test counters from the previous task, we added two more test counters, 6 and 9. Does your simulation test-

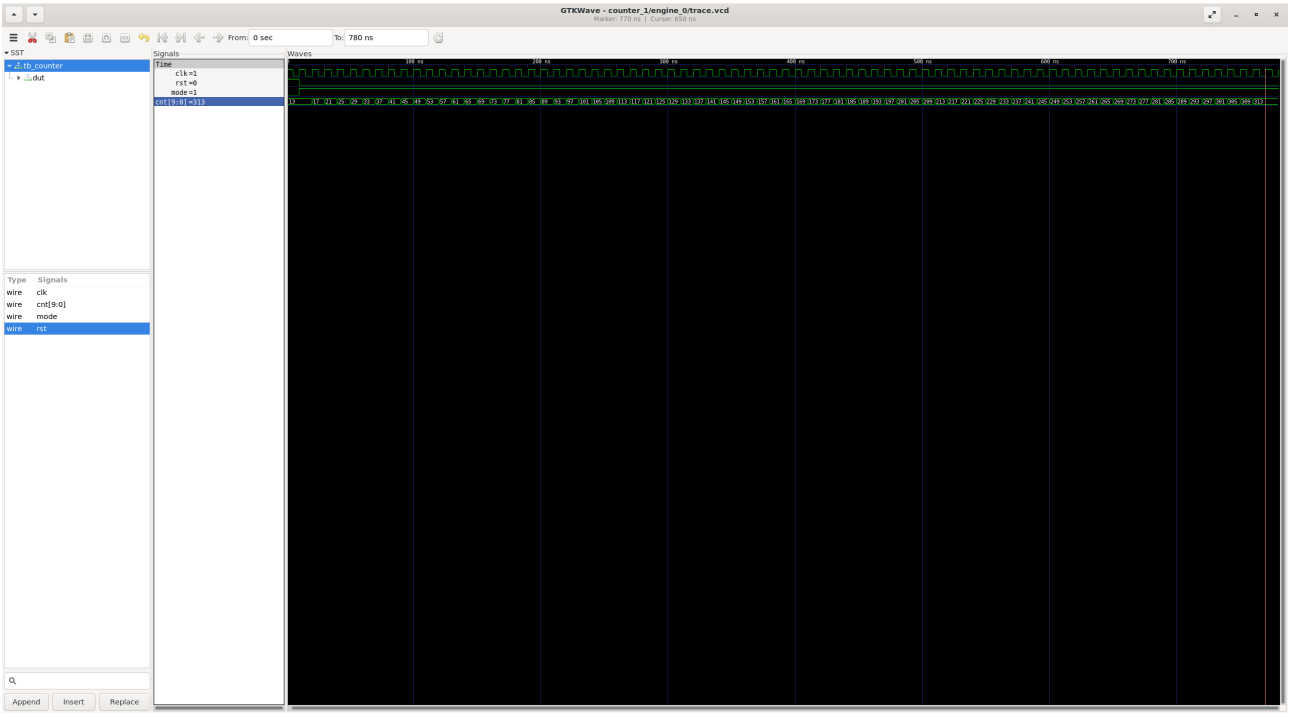

Figure 2: *GTKWave* showing the counterexample trace generated in Listing 6

Table 2: Functionality of test counters

Target	Functionality
0	correctly implemented counter
1	cnt may be greater than maximum
2	cnt may be less than minimum
3	cnt does not jump over invalid
4	increment is 1 larger than specified
5	cnt jumps (up) over invalid to value 1 larger than specified
6	increment may be 1 (other than 0 as specified) near maximum
7	decrement is 1 less than specified
8	cnt jumps (down) over invalid to value 1 smaller than specified
9	decrement may be 1 (other than 0 as specified) near minimum
10	initial value is 1 larger than specified

bench detect the bugs in them? Try it and copy your *tb_counter.v* from the previous task to this task's project directory (where the *Makefile* resides), and run `make 6` and `make 9`. Table 2 summarizes the test counters' functionality.

We also provide a skeleton *tb_counter.sv* in this ZIP archive. Your task is to extend this file with more assertions such that your testbench detects all bugs (verification task 0 must not fail, all other verification tasks must fail).

Check the properties using the command given in Listing 4, just like we did in Listings 5 and 6.

You will need a minimum of SystemVerilog for this task. Again, you may use the *if/else* statement to distinguish between cases. Also, the SystemVerilog *$past* statement will be helpful to solve your task. The web provides valuable resources about property checking with SVA.[2][3][4]

Submission details

To submit your formal testbench, attach your filled version of *tb_counter.sv* to your submission e-mail.

[2] https://yosyshq.readthedocs.io/projects/ap109/en/latest/
[3] https://symbiyosys.readthedocs.io/en/latest/verilog.html
[4] https://www.chipverify.com/systemverilog/systemverilog-assertions